The University of Melbourne
Department of Computing and Information Systems

Declarative Programming
COMP30020/COMP90048

Semester 2, 2016

Project Specification

Project due Monday, 12 September 2016 at 5pm
Worth 15%

The objective of this project is to practice and assess your understanding of functional programming and Haskell. You will write code to implement a logical deduction game.

The Game

For this project, you will implement a two-player logical guessing game. Two players face each other, each with a complete standard deck of western playing cards (without jokers). One player will be the *answerer* and the other is the *guesser*. The answerer begins by selecting some number of cards from his or her deck without showing the guesser. These cards will form the *answer* for this game. The aim of the game is for the guesser to guess the answer.

Once the answerer has selected the answer, the guesser chooses the same number of cards from his or her deck to form the *guess* and shows them to the answerer. The answerer responds by telling the guesser these five numbers as *feedback* for the guess:

1. How many of the cards in the answer are also in the guess (*correct cards*).
2. How many cards in the answer have rank lower than the lowest rank in the guess (*lower ranks*). Ranks, in order from low to high, are 2–10, Jack, Queen, King, and Ace.
3. How many of the cards in the answer have the same rank as a card in the guess (*correct ranks*). For this, each card in the guess is only counted once. That is, if the answer has two queens and the guess has one, the correct ranks number would be 1, not 2. Likewise if there is one queen in the answer and two in the guess.
4. How many cards in the answer have rank higher than the highest rank in the guess (*higher ranks*).
5. How many of the cards in the answer have the same suit as a card in the guess, only counting a card in the guess once (*correct suits*). For example, if the answer has two clubs and the guess has one club, or vice versa, the correct suits number would be 1, not 2.

Note that the order of the cards in the answer and the guess is immaterial, and that, since they come from a single deck, cards cannot be repeated in either answer or guess.

The guesser then guesses again, and receives feedback for the new guess, repeating the process until the guesser guesses the answer correctly. The object of the game for the guesser is to guess the answer with the fewest possible guesses.

A few examples of the feedback for a given answer and guess:

Answer	Guess	Feedback	Explanation
3♣,4♡	4♡,3♣	2,0,2,0,2	2 answer cards match guess cards, no answer cards less than a 3, 2 answer cards match guess ranks, no answer cards greater than a 4, 2 answer cards match guess suits.
3♣,4♡	3♣,3♡	1,0,1,1,2	1 exact match (3♣), no answer cards less than a 3, 1 answer card matches a guess rank, 1 answer card greater than a 3, 2 answer cards match guess suits.
3♢,3♡	3♠,3♣	0,0,2,0,0	No exact matches, no answer cards less than a 3, 2 answer cards match a guess rank, no answer cards greater than a 3, no answer cards match guess suits.
3♣,4♡	2♡,3♡	0,0,1,1,1	No exact matches, no answer cards less than a 2, 1 answer card matches a guess rank (the 3), 1 answer card greater than a 3, 1 answer card match a guess suit (♡).
A♣,2♣	3♣,4♡	0,1,0,1,1	No exact matches, 1 answer card less than a 3 (the 2; A is high), no answer card matches a guess rank, 1 answer card greater than a 4 (the A), 1 answer card match a guess suit (either ♣; you can only count 1 because there's only 1 ♣ in the guess).

The Program

For this assignment, you will write Haskell code to implement code for both the *answerer* and *guesser* parts of the game. This will require you to write a function to evaluate a guess to produce feedback, one to provide an initial guess, and one to use the feedback from the previous guess to determine the next guess. The latter function will be called repeatedly until it produces the correct guess. You will find it useful to keep information between guesses; since Haskell is a purely functional language, you cannot use a global or static variable to store this. Therefore, your initial guess function must return this game state information, and your next guess function must take the game state as input and return the updated game state as output. You may put any information you like in the game state, but you *must* define a type `GameState` to hold this information.

I will supply a `Card` module providing the `Card`, `Rank`, and `Suit` types and their constructors. This implementation has a few enhancements from the types presented in lectures. First, all three types are in the `Eq` class. All are also in the `Bounded` and `Enum` class, which means, for example, that `[minBound..maxBound]::[Card]` is the list of all cards in order, from 2♣to A♠, and similarly `[minBound..maxBound]::[Rank]` is the list of ranks from 2 to Ace, and similarly for `Suit`. This also means that, for example, `succ (Card Club R5) == (Card Club R6)` and `succ (Card Heart Ace) == (Card Spade R2)`. Read the documentation for `Bounded` and `Enum` classes for more things you can do with them.

The `Rank` and `Suit` types are also in the `Ord` class, so for example `Ace > King` and `Heart < Spade`. For convenience, all three types are also in the `Show` class so that ranks and suits are shown as a single character (10 is shown as `T`), and cards as two characters: rank followed by suit. All three are also in the Read class, which means that, for example, `(read "[2C,AH]")::[Card]` returns the list `[Card Club R2, Card Heart Ace]` (which would be printed as `[2C,AH]`).

You must define the following three functions, as well as the `GameState` type:

feedback :: [Card] → [Card] → (Int,Int,Int,Int,Int)
takes a target and a guess (in that order), each represented as a list of `Card`s, and returns the five feedback numbers, as explained above, as a tuple.

initialGuess :: Int → ([Card],GameState)
takes the number of cards in the answer as input and returns a pair of an initial guess, which should be a list of the specified number of cards, and a game state. The number of cards specified will be 2 for most of the test, and 3 or 4 for the remaining tests, as explained below.

nextGuess :: ([Card],GameState) → (Int,Int,Int,Int,Int) → ([Card],GameState)
takes as input a pair of the previous guess and game state, and the feedback to this guess as a quintuple of counts of correct cards, low ranks, correct ranks, high ranks, and correct suits, and returns a pair of the next guess and new game state.

You must call your (main) source file `Proj1.hs` or `Proj1.lhs`, and it must contain the module declaration:

```
module Proj1 (feedback, initialGuess, nextGuess, GameState) where
```

You may divide your code into as many files as you like, as long as your main file (and the files it imports) imports all the others. But do not feel you need to divide your program into many files if it is reasonably small.

I will also provide a test driver program called `Proj1test.hs` to allow you to conveniently test your `Proj1` implementation. You can specify the cards of the answer on the command line, and the program will print out each guess and the corresponding feedback and report the number of guesses needed to find the answer. The `Proj1test.hs` is similar to the test driver I will use for testing your code. You can compile and link your code for testing using the command:

```
ghc -O2 --make Proj1test
```

I will use a similar command to compile your code in preparation for testing.

Assessment

Your project will be assessed on the following criteria:

30% Quality of your code and documentation;

10% Correctness of your implementation of `feedback`.

20% Correctness of your implementation of `initialGuess` and `nextGuess` for 2-card targets

30% Quality of the guesses made by your implementation, as indicated by the number of guesses needed to find 2-card targets;

10% Correctness and quality of your guesses for 3- and 4-card targets.

The supplied `Proj1test` implementation will use the number of cards specified on the command line to determine the number to specify to the `initialGuess` function.

Note that timeouts will be imposed on all tests. You will have at least 10 seconds to guess each answer, regardless of how many guesses are needed. Executions taking longer than that may be unceremoniously terminated, leading to that test being assessed as failing. Your programs will be compiled with `ghc -O2` (optimisation enabled) before testing, so 10 seconds per test is a very reasonable limit. The same 10 second timeout will apply to the 3 and 4 card test cases.

See the Project Coding Guidelines on the LMS for detailed recommendations for coding style. These guidelines will form the basis of the quality assessment of your code and documentation, which is worth 30% of the project mark, so read them carefully.

Submission

The project submission deadline is Monday, 12 September 2016 at 5pm.

You must submit your project from one of the unix servers `nutmeg.eng.unimelb.edu.au` or `dimefox.eng.unimelb.edu.au`. Make sure the version of your program source files you wish to submit is on this host, and then `cd` to the directory holding your source code and issue the command:

```
submit COMP90048 proj1 Proj1.hs
```

If your code spans multiple source files, add the extra ones to the end of that command line.

Important: you must wait a few minutes after submitting, and then issue the command

```
verify COMP90048 proj1 | less
```

This will show you the test results from your submission, as well as the file(s) you submitted. If your program compiles and runs properly, you should see the line "`Running tests`", followed by forty test runs, each showing the number of guesses your program took. If you do not see test runs like this, then your program did not work correctly. If the test results show any problems, correct them and submit again. You may submit as often as you like; only your final submission will be assessed.

If you wish to (re-)submit after the project deadline, you may do so by adding "`.late`" to the end of the project name (*i.e.,* `proj1.late`) in the `submit` and `verify` commands. But note that a penalty, described below, will apply to late submissions, so you should weigh the points you will lose for a late submission against the points you expect to gain by revising your program and submitting again.

It is your responsibility to verify your submission.

Windows users should see the LMS Resources list for instructions for downloading the (free) Putty and Winscp programs to allow you to use and copy files to the department servers from windows computers. Mac OS X and Linux users can use the `ssh`, `scp`, and `sftp` programs that come with your operating system.

Late Penalties

Late submissions will incur a penalty of 0.5% of the possible value of that submission per hour late, including evening and weekend hours. This means that a perfect project that is

much more than 4 days late will receive less than half the marks for the project. If you have a medical or similar compelling reason for being late, you should contact the lecturer as early as possible to ask for an extension (preferably before the due date).

Hints

1. A very simple approach to this program is to simply guess every possible combination of different cards until you guess right. There are only $52 \times 51/2 = 1326$ possible answers, so on average it should only take about 663 guesses, making it perfectly feasible to do in 10 seconds. However, this will give a very poor score for guess quality.

2. A better approach would be to only make guesses that are consistent with the answers you have received for previous guesses. You can do this by computing the list of possible answers, and removing elements that are inconsistent with any answers you have received to previous guesses. A possible answer is inconsistent with an answer you have received for a previous guess if the answer you would receive for that guess and that (possible) answer is different from the answer you actually received for that guess. For two-card answers, the initial list of possible guesses is only 1326 elements, and rapidly shrinks with feedback, so this is quite feasible. This scales up to about 5 cards, when the initial number of possible guesses is 2,598,960. Beyond that, another strategy would need to be used, but you only need to handle 2 to 4 cards.

   You can use your `GameState` type to store the list of remaining possible answers, and pare it down each time you receive feedback for a guess.

3. The best results can be had by carefully choosing a guess that is likely to leave you the smallest remaining list of possible answers. You can do this by computing, for each remaining possible answer, the average number of possible answers it will leave if you guess it. Given a candidate guess G (which should be selected from the remaining possible answers), compute the feedback you will receive for each possible answer A if G is the guess and A is the answer. If you group all the As by the feedback they give you, your aim is to have many small groups, because that means if you make G your guess, that will probably leave few possible answers when you receive the feedback. Therefore the expected number of remaining possible answers for that guess is the average of the sizes of these groups, weighted by the sizes of the groups. That is, it is the sum of the squares of the group sizes divided by the sum of the group sizes.

4. For the first guess, when the list of possible answers is at its longest, you may want to use a different approach. Given the way the feedback works, the best first guess would be to choose two cards of different suits and with ranks about equally distant from each other and from the top and bottom ranks. In general, for an n card answer, you should choose ranks that are about $13/(n+1)$ ranks apart.

5. Note that these are just hints; you are welcome to use any approach you like to solve this, as long as it is correct and runs within the allowed time.

6. For a two card answer, with a good guessing strategy such as outlined above, 4 or 5 guesses is usually enough to guess it. Surprisingly, adding more cards does not increase the number of guesses needed very much.

Note Well:

This project is part of your final assessment, so cheating is not acceptable. Any form of material exchange between teams, whether written, electronic or any other medium, is considered cheating, and so is the soliciting of help from electronic newsgroups. Providing undue assistance is considered as serious as receiving it, and in the case of similarities that indicate exchange of more than basic ideas, formal disciplinary action will be taken for all involved parties. If you have questions regarding these rules, please ask the lecturer.